Claims

What Is Claimed Is

1) A method, called stega-cipher, comprising the step of using random keys in combination with steganographic technique to encode additional information into digitized samples in a manner that does not significantly degrade the signal generated from the sample stream and makes it impossible to extract the information without the keys

2)

An electronic, digital apparatus for stega-cipher encoding or decoding a message, represented as a series of data bits, into or out of a series of digitized samples comprising:

a) a high speed sample buffer, for holding and quickly accessing and transforming 128 digitized samples

b) a digital signal processor or microprocessor powerful enought to perform fast fourier transforms

c) a high speed memory, in the form of RAM or SRAM to contain information representing

1)128 bit primary mask

2)256 bit convolutional mask

3)1024 bit start of message delimeter

4)256 bit mask calculation buffer

5)2 Kilobyte message buffer

6)an integer representing a message bit index

7)an integer respresenting message size

8)an integer representing an index into the primary mask

9)an integer representing an index into the convolutional mask

10)an integer representing the state of the decode process

11 a boolean truth table representing the stega-cipher map function

12) a boolean flag indicating a complete message has been decoded, or encoded

13) an integer representing the number of samples to read into the sample buffer

14) a boolean flag indicating the size of a message has been decoded

d) a means to acquire digital samples from an attached digital circuit or analog input

e) a means to output digital samples to an attached digital circuit or analog output

f) a means to recieve the values of [c1] - [c5] and [c11] from an attached digital circuit

e) a means to output the message stored in [c5] as the result of a decode process and the value of [c11] to an attached digital circuit

f) at least one data bus to transfer information from [d] to [a]

[a] to [b]

[b] to [a]

[a] to [e]

[f] to [c]

[c] to [e]

g) a clock signal to drive [b] and control the operation of the circuit

3)

a method for utilizing the apparatus of claim 2 to encode stega-cipher information into a sample stream of data. wherein said method is comprised of the following steps

I. Initialization

Generate a mask set to be used for encoding

the set includes a random 128 bit primary mask

a random 256 bit convolution mask

a random 1024 bit start of message delimeter

Obtain a message to be encoded

compress and encrypt the message if desired

Generate a message bit stream to be encoded such that the stream is comprised of

1024 bits of start of message delimeter

32 bits representing the number of message bytes to follow

{ the bytes of the message, optionally compressed and/or encrypted }

in that order

load the message bit stream , the stega-cipher map truth table, the primary mask, the convolution mask and the start of message delimeter into [c]

the processor sets the primary and convolution mask indexes to 0, and sets the message bit index to 0, and sets the message size integer equal to the total number of bits in the message bit stream

the processor sets the message encoded flag to false

II. the processor reads a window of 128 samples from the sample input means and stores them sequentially in the sample buffer

if this step fails, the process ends

III. The processor sets the primary mask index to 0 and uses it to loop through the sample buffer from the first sample to the last sample, incrementing the index each time, visiting each sample. For each sample position, the processor computes the value of the mapping function, which is either true or false. To do this, the processor uses the bit of the primary mask representing the current sample (0-127) and the bit of the convolution mask indicated by the convolution index (0-255) to calculate an offset into the truth table representing the

mapping function. The processor then obtains the bit value stored there. If the value is true (1), the processor encodes the bit of the message indicated by the message bit index into the current sample and increments the message bit index. If at any time during this loop the message bit index equals the total message bits, then the processor performs step I. sets the message encoded flag and exits the loop.

IV. The processor outputs the samples in the sample buffer via the sample output means. If the message encoded flag is set, the processor turns the flag off, and jumps back to step II.

V. The processor increments the convolution index. If the convolution index equals the length of the convolution mask in bits (256), then the processor sets the convolution index to 0. Finally, the processor jumps back to step II.

4) a method for utilizing the apparatus of claim 2 to encode stega-cipher information into a sample stream of data wherein said method is comprised of the following steps

I. Initialization

Generate a mask set to be used for encoding

the set includes a random 128 bit primary mask

a random 256 bit convolution mask

a random 1024 bit start of message delimeter

Obtain a message to be encoded

compress and encrypt the message if desired

Generate a message bit stream to be encoded such that the stream is comprised of

1024 bits of start of message delimeter

32 bits representing the number of message bytes to follow

{ the bytes of the message, optionally compressed and/or encrypted }

in that order

load the message bit stream , the stega-cipher map truth table, the primary mask, the convolution mask and the start of message delimeter into [c]

the processor sets the primary and convolution mask indexes to 0, and sets the message bit index to 0, and sets the message size integer equal to the total number of bits in the message bit stream

II. the processor reads a window of 128 samples from the sample input means and stores them sequentially in the sample buffer

if this step fails, the process ends

IIA. The processor computes the Fast Fourier Transform of the samples in the message buffer, replacing them with spectral sample values.

III. The processor uses the primary mask index to loop through the sample buffer from the first sample to the last sample, visiting each one. For each sample position, the processor computes the value of the mapping function, which is either true or false. To do this, the processor uses the bit of the primary mask representing the current sample (0-127) and the bit of the convolution mask indicated by the convolution phase (0-255) to calculate an offset into the truth table representing the mapping function. The processor then obtains the bit value stored there. If the value is true, the processor encodes the bit of the message indicated by the message bit index into the current sample and increments the message bit index. If at any time during this loop the message bit index equals the total message bits, then the processor performs step I, sets the message encoded flag and exits the loop.

IV. The processor outputs the samples in the sample buffer via the sample output means

IVA. The processor computes the inverse Fast Fourier Transform of the spectral sample values in the sample buffer, converting them back to amplitude samples. If the message encoded flag is on, the processor turns it off and jumps to step II.

V. The processor increments the convolution index. If the convolution index equals the length of the convolution mask in bits (256), then the processor sets the convolution index to 0. Finally, the processor jumps back to step II.

5) The method of claims 3 or 4 where the encoding of the message bit into the sample in step [illegible] consists of altering the [illegible] significant bit of the sample to match the message bit

6) The method of claims 3 or 4 where the encoding of the message bit into the sample in step III consists of altering the sample value so that it falls within a specified range of values relative to its orginial value. For instance, if the message bit is 0, the sample value must have a least significant tens digit of 0-4, and if the message bit is 1, the sample value must have a least significant tens digit of 5-9. If the message bit is one, and the original sample value is 1003, then the sample value could be changed to [995-999] or [1005-1009] in order to signify a 1 message bit. If the message bit were 0 under this scheme, the sample could assume any value [1000-1004] or could be left as is.

7)

a method for utilizing the apparatus of claim 2 to decode stega-cipher information from a sample stream of data, wherein said method is comprised of the following steps

I. Initialization

Obtain a mask set to attempt decoding with

the set includes a 128 bit primary mask

a 256 bit convolution mask

a 1024 bit start of message delimeter

load the stega-cipher map truth table, the primary mask, the convolution mask and the start of message delimeter into [c]

the processor sets the primary and convolution mask indices to 0, and sets the message bit index to 0, and sets the message size integer equal to 0

the processor sets the message decoded flag to false

the processor sets the state of the decode process to UNSYNCHRONIZED

II. The processor checks the state of the process. If it is UNSYNCHRONIZED, the processor sets the number of samples to read equal to 1 and sets the convolution index to 0. Otherwise, the processor sets the number of samples to read equal to 128

III. The processor reads the number of samples specified in [c13] and stores them in the sample buffer. If this step fails the process ends, with no message decoded.

IV. The processor sets the primary mask index to 0 and uses it to loop through the sample buffer from the first sample to the last sample, incrementing the index each time, visiting each sample. For each sample position, the processor computes the value of the mapping function, which is either true or false. To do this, the processor uses the bit of the primary mask representing the current sample (0-127) and the bit of the convolution mask indicated by the convolution phase (0-255) to calculate an offset into the truth table representing the mapping function. The processor then obtains the bit value stored there. If the value is true, the processor decodes the bit of the message indicated by the message bit index into the current sample, stores the bit into the message buffer at the message bit index and increments the message bit index.

V. If the state is UNSYNCHRONIZED or SYNCHRONIZING, the processor compares the decoded bits in the message buffer to the start of message delimeter. If the number of bits in the message buffer is less than the number of bits in the start of message delimeter and the bits match, the processor sets the state to SYNCHRONIZING. Otherwise, if the first 1024 bits in the message buffer match the start of message delimeter the processor sets the state to SYNCHRONIZED. Otherwise, the processor sets the state to UNSYNCHRONIZED.

VI. If the state is SYNCHRONIZED and there are at least 1056 bits in the message buffer, the processor sets the state to SYNCH_AND_SIZE and copies bits 1024-1055 of the message buffer to the message size container

VII. If the state is SYNCH_AND_SIZE and the (number of bits in the message buffer - 1056) divided by 8 is greater than or equal to the message size then a complete message is in the message buffer and the process ends

VIII. The processor increments the convolution index. If the convolution index equals the number of bits in the convolution mask, the processor sets the convolution index to 0. The processor jumps to step II.

8)

a method for utilizing the apparatus of claim 2 to decode stega-cipher information from a sample stream of data, wherein said method is comprised of the following steps

I. Initialization

Obtain a mask set to attempt decoding with

the set includes a 128 bit primary mask

a 256 bit convolution mask

a 1024 bit start of message delimeter

load the stega-cipher map truth table, the primary mask, the convolution mask and the start of message delimeter into [c]

the processor sets the primary and convolution mask indices to 0, and sets the message bit index to 0, and sets the message size integer equal to 0

the processor sets the message decoded flag to false

the processor sets the state of the decode process to UNSYNCHRONIZED

II. The processor checks the state of the process. If it is UNSYNCHRONIZED, the processor sets the number of samples to read equal to 1 and sets the convolution index to 0. Otherwise, the processor sets the number of samples to read equal to 128

III. The processor reads the number of samples specified in [c13] and stores them in the sample buffer. If this step fails the process ends, with no message decoded.

IIIa. The processor computes the Fast Fourier Transform of the samples in the sample buffer and replaces them with frequency domain samples.

IV. The processor sets the primary mask index to 0 and uses it to loop through the sample buffer from the first sample to the last sample, incrementing the index each time, visiting each sample. For each sample position, the processor computes the value of the mapping function, which is either true or false. To do this, the processor uses the bit of the primary mask representing the current sample (0-127) and the bit of the convolution mask indicated by the convolution phase (0-255) to calculate an offset into the truth table representing the mapping function. The processor then obtains the bit value stored there. If the value is true, the processor decodes the bit of the message indicated by the message bit index into the current sample, stores the bit into the message buffer at the message bit index and increments the message bit index.

V. If the state is UNSYNCHRONIZED or SYNCHRONIZING, the processor compares the decoded bits in the message buffer to the start of message delimeter. If the number of bits in the message buffer is less than the number of bits in the start of message delimeter and the bits match, the processor sets the state to SYNCHRONIZING. Otherwise, if the first 1024 bits in the message buffer match the start of message delimeter the processor sets the state to SYNCHRONIZED. Otherwise, the processor sets the state to UNSYNCHRONIZED.

VI. If the state is SYNCHRONIZED and there are at least 1056 bits in the message buffer, the processor sets the state to SYNCH_AND_SIZE and copies bits 1024-1055 of the message buffer to the message size container.

VII. If the state is SYNCH_AND_SIZE and the (number of bits in the message buffer - 1056) divided by 8 is greater than equal to the message size then a complete message is in the message buffer and the process ends

VIII. The processor increments the convolution index. If the convolution index equals the number of bits in the convolution mask, the processor sets the convolution index to 0. The processor jumps to step II.

9) The method of claims 7 or 8 where the decoding of the message bit from the sample in step III consists of reading least significant bit of the sample.

10) The method of claims 7 or 8 where the decoding of the message bit from the sample in step III consists of mapping a range of values onto a message bit. For instance, if the sample value has a least significant tens digit of 0-4, then the message bit decoded is 0 and if the value has a least significant tens digit of 5-9, the message bit is 1.

11)

the methods or claims 4 and 8 where the boolean truth table is TRUE for all inputs. That is f(bit1,bit2) = TRUE for all values of bit1 and bit2

12) the methods of claims 3,4,5,6,7,8,9,10 and 11 where the samples are obtained from a sample stream representing digitized sound or music

13) the method of claim 12 where the identical encode or decode process is performed on two sample streams representing channel A and channel B of digitized stereo sound

14) the method of claim 12 where the sample streams representing channel A and channel B of a digitized stereo sound are interleaved before input to the stega-cipher apparatus and algorithm as a single sample stream and separated into two channels upon output.

15) the methods of claims 3,4,5,6,7,8,9,10 and 11 where the samples are obtained from a sample stream representing digitized video

16) the methods of claims 3,4,5,6,7,8,9,10 and 11 where the samples are obtained from a sample stream representing a digitized image.

17)
the apparatus of claim 2 where said apparatus is contained in tamper-resistant packaging, which destroys circuitry and information stored in said apparatus if broken

18) The method of claims 3 and 7 where a pre-encoding step is used which further customizes the message to be encoded by first calculating over which exact windows in the sample stream a message will be encoded, then computing a secure one way hash function of the samples in those windows, where the hash function is designed to be insensitive to

changes in the samples induced by the stega-cipher apparatus of claim 1 and the methods of claims 3 and 7, and then including the resulting hash values in the message, which is signed and encrypted and finally encoded into the samples.

This method is accomplished by 1) calculating the size of the message plus the added hash value, which adds a constant, predicatable number of bytes to the message 2) calculating the size of a signed encrypted version of the message 3) processing the sample stream in a manner which emulates the calculations performed in the method of claims 2 and 6 for the purpose of calculating hash values of each series of windows that will be used to encode the message, and creating a modified copy of the message which matches each particular series uniquely with each hash value, and 4) feeding this message to the apparatus of claim 1 as decribed in claims 3 and 7.

19) the methods of claims 3,4,5,6,7,8,9,10,11,12,13,14 and 18 where a trusted authority for the online distribution of content encodes the following information into a sample stream using the apparatus of claim 2

the title

the artist

the copyright holder

the body to which royalties should be paid

general terms for publishers' distribution

20) the method of claim 19 where a trusted authority combines the information listed in claim 19 and additionally a secure private key signed message from a publisher containing the following information

the title

the publisher's identification

the terms of distribution

any consideration paid for the right to distribute the content

a brief statement of agreement with all terms listed above

and signs and encrypts this combined message using a public key cryptosystem and further encodes this signed and encrypted message into a sample stream

21) the method of claim 20 where a publisher obtains the sample stream encoded in the claim 20 and additionally obtains the information listed in claim 19 from the authority and combines this with a message recieved from a consumer, which has been signed using a public key cryptosystem and where the signed message contains the following information

the content title

consumer identification

the terms of distribution

the consideration paid for the content

a brief statement of agreement with the terms above

and further, the publisher uses a public key cryptosystem to sign this combined information and finally encodes the signed information as a stega-cipher watermark

22) the method of claims 3,4,5,6,7,8,9 and 10 where the sample stream is obtained from at least one audio track contained within a digitized movie, video gae software, or other software.

23) the method of claims 3,4,5,6,7,8,9 and 10 where the sample stream is obtained from at least one digitized movie or still image contained within video game or other software.

24) the method of encoding information into content where the information is contained in the differences between samples, rather than the value of an individual sample

add B3

add C4

Appendix - Psuedo-code

```
const int WINDOW_RESET = 256;
const int WINDOW_SIZE = 128;
const int MARKER_BITS = 1024;
const int CHUNK_BITS = 2048 * 8;

int window_offset;
int msg_bit_offset;
int frequency_offset;
Boolean useCell;

/* 8 bits per bye, 1 byte per char */
unsigned char frequency_mask[WINDOW_SIZE/8];
unsigned char window_mask[WINDOW_RESET/8];
unsigned char msg_start_marker[MARKER_BITS/8];
unsigned char msg_end_marker[MARKER_BITS/8];
Int16 amplitude_sample_buffer[WINDOW_SIZE];
float power_frequency_buffer[WINDOW_SIZE];
unsigned char message_buffer[CHUNK_BITS/8];

void doFFT(Int16 *amp_sample_buffer, float *power_freq_buffer,int size);
void doInverseFFT(Int16 *amp_sample_buffer, float *power_freq_buffer,int size);
void initialize();
Bit getBit(unsigned char *buffer,int bitOffset);
Boolean map(Bit window_bit, Bit band_bit, int window, int frequency);
```

```
Boolean getSamples(Int16 *amplitude_sample_buffer,int samples);
void encode()

void initialize()
{
        /* message to be encoded is generated */
        /* message is prefixed with 1024 bit msg_start_marker */
        /* message is suffixed with 1024 bit msg_end _marker */
        /* remaining space at end of message buffer padded with random bits */
        window_offset = 0;
        msg_bit_offset = 0;
        frequency_offset = 0;
        frequency_mask loaded
        window_mask loaded
        zeroAmpSampleBuffer();
}

Boolean getSamples(Int16 *buffer,int samples)
{
        /* get samples number of samples and shift them contiguously into the sample
           buffer from right to left*/
        if(samples < samples available)
                return false;
        else
                return true;
}
```

```
void doFFT(Int16 *sample_buffer, float *spectrum_buffer, int size)
{
        calculate FFT on sample_buffer, for size samples
        store result in spectrum buffer
}

void doInverseFFT(Int16 *sample_buffer,float *spectrum_buffer,int size)
{
        calculate inverse FFT on spectrum_buffer
        store result in sampe_buffer
}

Bit getBit(unsigned char *buffer,in bitOffset)
{
        returns value of specified bit in specified buffer
        either 0 or 1, could use Boolean (true/false) values for bit set of bit off
}

Boolean map(Bit window_bit,Bit band_bit,int window, int frequency_
{
        /* this is the function that makes the information difficult to find */
        /* the inputs window_bit and band_bit depend only on the mask values
                used for encoding the information, they are 1) random, 2) secret */
        /* window and frequency values are used add time and frequency band dependent
                complexity to this function */
        /* this function is equivalent to a Boolean truth table with window * frequency * 4
                possible input combinations and 2 possible output */
```

```
        /* for any input combination, the output is either true or false */
        /* window ranges from 0 to WINDOW_RESET -1
        /* frequency ranges from 0 to WINDOW_SIZE - 1 */
        return calculated truth value
    }

void encodeBit(float *spectrum_buffer,int freq_offset,Bit theBit)
{
        /* modifies the value of the cell in spectrum_buffer, indexed by freq_offset
                in a manner that distinguishes each of the 2 possible values of theBit,
                1 or 0
        */
        /* suggested method of setting the Least Significant bit of the cell == theBit */
        /* alternative method of rounding the value of the cell upward or downward to
                certain fractional values proposed
                i.e. <= .5 fractional remainder signifies 0, > .5 fraction remainder
                        signifies 1
        */
}

void encode()
{
        initialize();

        do {

        if(getSamples(amplitude_sample_buffer) == false)
```

```
        return

    doFFT(amplitude_sample_buffer,power_frequency_buffer,WINDOW_SIZE);

    for (frequency_offset = 0; frequency_offset < WINDOW_SIZE;
    frequency_offset++){

        useCell = map(getBit(window_mask,window_offset),
                    getBit(frequency_mask,frequency_offset),
                    window_offset, frequency_offset);

        if(useCell == true){
            encodeBit(power_frequency_buffer,frequency_offset,
                    getBit(message_buffer,msg_bit_offset));
            message_bit_offset ++;
            if(msg_bit_offset == MESSAGEBITS){
                initialize();
                break; /* exit frequency loop */
            }
        }
    }

    doInverseFFT(amplitude_sample_buffer,power_frequency_buffer,
        WINDOW_SIZE);

    outputSamples(amplitude_sample_buffer);
```

```
        window_offset++;
        if(window_offset == WINDOW_RESET){
                window_offset = 0;
        }


    } while(true);
}
```

The encode() procedure processes an input sample stream using the specified frequency and window masks as well as a pre-formatted message to encode.

encode() processes the sample stream in windows of WINDOW_SIZE samples, contiguously distributed in the sample stream, so it advances WINDOW_SIZE samples at a time.

For each sample window, encode() first compute the FFT of the window, yielding its Power Spectrum Estimation. For each of these window PSEs, encode() then uses the map() function to determine where in each PSE to encode the bits of the message, which it reads from the message buffer, on ebit at a time. Each time map() returns true, encode() consumes another sample from the message.

After each window is encoded, encode() computes the inverse FFT on the PSE to generate a modified sample window, which is then output as the modified signal. It is important the sample windows NOT overlap in the sample stream, since this would potentially damage the preceeding encoding windows in the stream.

Once the message is entirely encoded, including its special end of message marker bit stream, encode() resets it internal variables to begin encoding the message once more in the next window. encode() proceeds in this manner until the input sample stream is exhausted.

```
enum {
        Synchronizing,
        Locked
}; /* decode states */

unsigned char message_end_buffer[MARKER_BITS];

Bit decodeBit(float *spectrum_buffer,int freq_offset)
{
        /* reads the value of the cell in spectrum_buffer, indexed by freq_offset
                in a manner that distinguishes each of the 2 possible values of an
                encoded bit, 1 or 0
        */
        /* suggested method of testing the Least Significant bit of the cell */
        /* alternative method of checking the value of the cell versus certain fractional
                remainders proposed.
                i.e. <= .5 fractional remainder signifies 0, > .5 fraction remainder
                        signifies 1
        */
        return either 1 or 0 as appropriate
}
```

```
Boolean decode()
{
        /* Initialization */
        state = Synchronizing
        window_offset = 0;
        set frequency mask
        set window mask
        clear sample buffer
        int nextSamples = 1;
        int msg_start_offset = 0;
        clear message_end_buffer
        Bit aBit;
        Boolean bitsEqual;

        do {

                if(state == Synchronizing){
                        nextSamples = 1;
                        window_offset = 0;
                }
                else
                        nextSamples = WINDOW_SIZE;

                if(getSamples(amplitude_sample_buffer) == false)
                        return false;

                doFFT(amplitude_sample_buffer,power_frequency_buffer,
```

```
WINDOW_SIZE);/* 2 */

for (frequency_offset = 0; frequency_offset < WINDOW_SIZE;
frequency_offset++){

    useCell = map(getBit(window_mask,window_offset),
            getBit(frequency_mask,frequency_offset),
            window_offset, frequency_offset);

    if(useCell == true){
        aBit = decodeBit(power_frequency_buffer,
                    frequency_offset);
        setBit(message_buffer,message_bit_offset,aBit);
        message_bit_offset ++;
    }
    else
        continue;
    if(state == Synchronizing){
        bitsEqual =
        compareBits(message_start_marker,message_buffer,
                message_bit_offset);
        if(!bitsEqual){
                message_bit_offset = 0;
                misaligned = true;
                break; /* exit frequency loop */
        }
        else if (message_bit_offset == MARKER_BITS)
```

```
                                    state == Locked;
                }
                else {
                        /* locked onto encoded stream */
                        shift aBit into right side of message_end_buffer
                        bitsEqual = compareBits(message_end_buffer,
                                msg_end_marker,MARKER_BITS),
                        if(bitsEqual)
                                return true;
                }
        }
    } while (true);
}
```

The decode() procedure scans an input sample stream using specified window and frequency masks, until it either decodes a valid message block, storing it in a message buffer, or exhausts the sample stream.

The decode() procedure starts in state Synchronizing, in which it does not know where in the sample stream the encoding windows are aligned. The procedure advances the sample window through the sample stream one sample at a time, performing the FFT calculation on each window, and attempting to decode valid message bits from the window. As it extracts each bit using the map() function, the decode() procdure compares these bits against the start of message marker. As soon as a mismatch is detected, the decode() procedure knows it is not yet properly aligned to an encoding window, and immediately ceases decoding bits from the current window and moves to the next window, offset by 1 sample. The decode() procedure continues in

this manner until it matches successfully the complete bitstream of a start of message marker. At this point the decode() procedure assumes it is aligned to an encoded message and can then decode bits to the message buffer quickly, advancing the sample window fully at each iterations. It is now in Locked mode. For each bit it stores in the message buffer when in Locked mode, the decode() procedure also shifts the same bit value into the least significant bit of the message_end_buffer. After each bit is decoded in Locked mode, the decode() procedure checks compares the message_end_buffer with the msg_end_marker in a bit by bit manner. When a complete match is found, decode() is finished and returns true. If the sample stream is exhausted before this occurs, decode() returns false. If decode() returns true, a valid message is stored in the message buffer, including the start and end of message markers.